Node Express CRUD using TypeScript and Dependency Injection

Framework used: Express Js v4.17.1  
Server: Node js v14.15.5

Language: TypeScript v4.2.4  
Query builder: Knex v0.95.6

Dbms: MySQL

GitHub Link for Project:

Start by creating a new directory where you keep your projects in your local development environment.

Let us assume we are inside **E:/NodeExamples.**  
In this directory we have created a project folder named **KnexCrudApp.**  
Inside project directory, use npm’s initializer command to create a **package.json** file:

**npm init –yes**  
  
The **--yes** flag uses the default settings when initializing a **package.json**

The **package.json** file created might look something like this:

{

"name": "KnexCrudApp",

"version": "0.0.1",

"description": "",

"main": "index.js",

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1"

},

"keywords": [],

"author": "",

"license": "MIT"

}

Now let us add an express package. From the terminal window, run the command:  
**npm install express -s**

Next, create a new file called **index.js** at the root of the project with the following code to trigger a minimal server:

const express = require('express');

const app = express();

const PORT = 8000;

app.listen(PORT, () => {

console.log(`[server]: Server is running http://localhost:${PORT}`);

});

**Now we will add TypeScript in the project**

Let’s add two libraries to the development server as **devDependencies**.

* **typescript** is a core library that helps to compile the TypeScript code to valid JavaScript
* **ts-node** is a utility library that helps to run a development server written using TypeScript directly from the terminal

To install them, from a terminal window run the following command:  
**npm install –D typescript ts-node**The **-D** flag is also known as **--dev** flag and is a specification for the package manager to install these libraries as **devDependencies**.  
Once these libraries are installed, go to the **package.json** file and see a new **devDependencies** object:

"devDependencies": {

"ts-node": "10.0.0",

"typescript": "4.2.4"

}

Install declaration files for Node Js and Express:

Declaration files are predefined modules that describe the shape of JavaScript values (the types present) for the TypeScript compiler. Type declarations are usually contained in files with a **.d.ts** extension. These declaration files are available for all libraries that are originally written in JavaScript and not TypeScript.  
To add these types or the declaration files related to a particular library or a module, you have to look for the packages that start with **@types** namespace.

For example, the type definitions for Express library is kept under a specific package called [**@types/express**](https://www.npmjs.com/package/@types/express). For using a utility library such as **bodyParser** (which is a middleware to parse an incoming request’s body), there is a specific type of definition module called [**@types/body-parser**](https://www.npmjs.com/package/@types/body-parser).  
  
To install type definitions for Node.js and Express, run the below command. Do note that, these type definitions are installed as **devDependencies**:

**npm** **i -D typescript @types/bcrypt @types/body-parser @types/cookie-parser @types/cors @types/express @types/jsonwebtoken @types/node**

Next, create a **tsconfig.json** file at the root of the development server project.  
We can create this file manually or we can use the following command to create it:

**tsc** -**init**

This file allows you to customize TypeScript configuration and add other configurations to compile the TypeScript project:  
{

"compilerOptions": {

"target": "es6",

"module": "commonjs",

"rootDir": "./",

"outDir": "./build",

"esModuleInterop": true,

"strict": true

}

}

The **compilerOptions** is a mandatory field that needs to be specified. The options used in the config above are:

* **target** allows us to specify the target JavaScript version that compiler will output
* **module** allows us to use a module manager in the compiled JavaScript code. The **commonjs** is supported and is a standard in Node.js
* **rootDir** is an option that specifies where the TypeScript files are located inside the Node.js project
* **outDir** specifies where the output of the compiled is going to be located
* **esModuleInterop** allows us to compile ES6 modules to **commonjs**modules
* **strict** is an option that enables strict type-checking options

There are other configuration options also, that you can add on for the TypeScript compiler but these are the basic configuration options specified that can help you to get started.

**Create an Express Server with .ts extension**Now you can easily convert the **index.js** to **index.ts** file. That is the first step. Rename the file to **index.ts.**

Open **index.ts** file. You can now use the **import** statements from ES6. The only required package right now in the **index.ts** file is **express.** Replace it with the following statement:  
  
import express from 'express';

// rest of the code remains same

const app = express();

const PORT = 8000;

app.listen(PORT, () => {

console.log(`[server]: Server is running http://localhost:${PORT}`);

});

The TypeScript compiler will handle the conversion of import statements to require statements.

## **Watching file changes with nodemon**

Another development related utility library we will use when working on Node.js projects is **[nodemon](https://www.npmjs.com/package/nodemon" \t "_blank).** Let’s install this using the command below:  
**npm** **install –g nodemon**

**nodemon** is a tool that helps develop Node.js based applications by automatically restarting the Node application when file changes in the directory are detected. To use it, you may add a **start** script in the **package.json** file as specified below:

"scripts": {

"start": "nodemon index.ts",

},

Now, open terminal, and run **npm start**.

The **ts-node** utility checks for any file changes in the current TypeScript project. If there is a TypeScript error, this module will let the **nodemon** crash the server and instead display that error.

## **Compile a TypeScript project**

To compile a TypeScript project to a valid JavaScript one, start by declaring a new script called build inside the **package.json** file:

"scripts": {

"build": "tsc --project ./",

},

TypeScript provides a command to compile the code called **tsc**. This command demands a flag to specify as to what to compile. The **--project** (shorthand: **-p**) is used to specify the project directory that the compiler can pick the code files from to compile to valid JavaScript. The **./** specifies the root project.

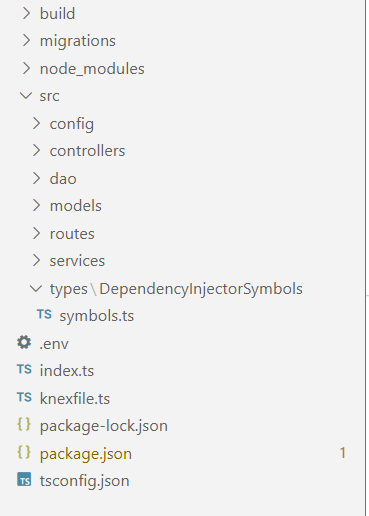
From the terminal window, run the build command to compile the code:

**npm run build**

There is a new **build** directory created after this command executes successfully. Inside this directory, there is the TypeScript code compiled to valid JavaScript.

## **Directory Structure of the project**

Create the following directory structure in KnexCrudApp project



**build** and **migrations** folders will be generated automatically upon compiling typescript and creating migrations respectively.

Let us start by creating **models** folder. This folder will contain all the models to be used in the project. Create a file **Employee.ts** in models folder with following code in it:

export interface Employee{

    id?:number;

    fname:string;

    lname: string;

    phone: string;

    dob:Date;

    salary:number;

    createdAt?:Date;

    updatedAt?:Date;

}

Here **?** represents the optional attribute. It is not mandatory to fill id, createdAt, and updatedAt attributes because they will be autofilled by the query builder.

Now install **knex** query builder to perform CRUD operations on our model.

**npm** **install knex --save**

then install suitable database driver (for this example we are installing mysql driver)  
**npm** **install mysql**

In next step we will configure knex to connect with mysql. Define connection parameters in **.env** file.  
Create **.env** file in root directory with following key value pairs:  
DB\_HOST=localhost

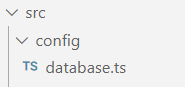
DB\_USER=root

DB\_PASS=

DB\_NAME=yoyodb

DB\_DIALECT=mysql

To configure knex with mysql create **database.ts** file in **config** directory.



In **database.ts** load **.env** file to use its parameters. To load .env file install a dependency **dotenv**

**npm** **install dotenv**

Do the following configs in **database.ts**

import \* as dotenv from 'dotenv';

dotenv.config();

export const DbConfig = {

    client: process.env.DB\_DIALECT,

    connection: {

        host: process.env.DB\_HOST,

        user: process.env.DB\_USER,

        password: process.env.DB\_PASS,

        database: process.env.DB\_NAME

    }

}

Now install knex CLI:

**npm** **install –g knex**

Now with the help of knex cli, generate a file to provide knex the db configurations which we have created recently.

Use following command to create it:

**knex init –x ts**

This will generate **knexfile.ts :**

This file will configure the knex with development, testing, and production environment. Here we can mention different set of config values for different environments. As of now we are configuring only the development environment. Database configs are already present in **./src/config/database.ts.** We have to import this to use. **knexfile.ts** will have following content:

import {DbConfig} from './src/config/database';

module.exports = {

  development: DbConfig

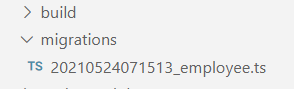
};

Here we have configured only development environment but, we can also configure other environments using attributes production: & test:

As of now, we are done with the db configuration part. Now let us create migration using knex cli:

**knex migrate:make employee –x ts**

This will create migrations directory in project root folder and in that folder you will find a migration file for the employee.



This migration file will contain the schema of the table.

import { Knex } from "knex";

export async function up(knex: Knex): Promise<any> {

   return knex.schema.createTable('employees', function (table) {

        table.increments();

        table.string('fname').notNullable();

        table.string('lname').notNullable();

        table.string('phone',15).notNullable().unique();

        table.date('dob').notNullable();

        table.float('salary').notNullable();

        table.timestamps();

    })

}

export async function down(knex: Knex): Promise<void> {

}

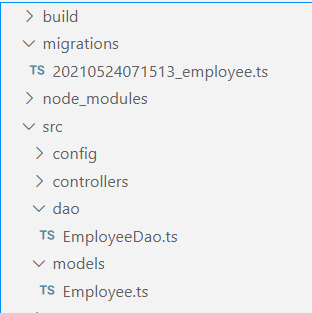
Now perform migration with the help of following command:

**knex migrate:up**

This will migrate the employees table to the configured database.

**Create Dao layer for CRUD operations**

This layer is responsible for performing CRUD related operations using knex. Create a dao file for employee as **EmployeeDao.ts** in dao directory



Content of **EmployeeDao.ts:**

import {Employee} from '../models/Employee';

import { DbConfig } from '../config/database';

import knex from "knex";

const db=knex(DbConfig);

export class EmployeeDao{

    async findAll(): Promise<Employee[]>{

        try {

            const employees:Employee[]=await db<Employee>('employees').select();

            return employees;

        } catch (error) {

            throw error;

        }

    }

    async find(limit: number,offset: number): Promise<Employee[]>{

        try {

            const employees:Employee[]= await db<Employee>('employees').select().limit(limit).offset(offset);

            return employees;

        } catch (error) {

            throw error;

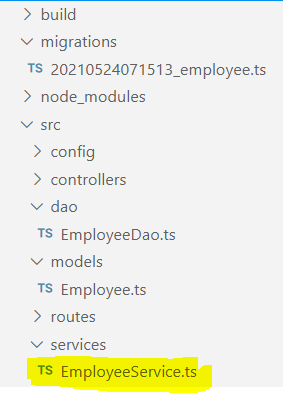
        }

    }

}

**Create Service layer**

This layer is responsible for executing business logics. Create a service for employee as **EmployeeService.ts** in services directory



Content of **EmployeeService.ts:**

import { EmployeeDao } from "../dao/EmployeeDao";

import { Employee } from "../models/Employee";

export class EmployeeService{

    private employeeDao: EmployeeDao=new EmployeeDao();

    async findAll(): Promise<Employee[]> {

        try {

            const employees:Employee[] = await this.employeeDao.findAll();

            return employees;

        } catch (error) {

            throw error;

        }

    }

    async find(pageNo: number, limit: number): Promise<Employee[]> {

        try {

            const offset:number = (pageNo - 1) \* limit;

            const employees:Employee[] = await this.employeeDao.find(limit, offset);

            return employees;

        } catch (error) {

            throw error;

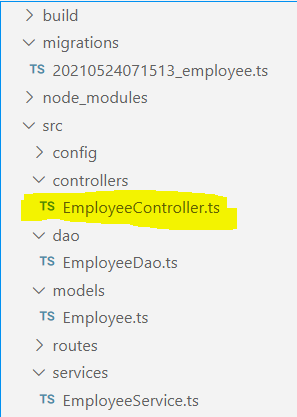
        }

    }

}

**Create Controller layer**

This layer is responsible for handling request and response. Create a controller for employee as **EmployeeController.ts** in controllers directory.



Content of **EmployeeController.ts**

import { Request, Response } from 'express';

import { Employee } from '../models/Employee';

import { EmployeeService } from '../services/EmployeeService';

export class EmployeeController {

    private employeeService: EmployeeService;

    constructor() {

        this.employeeService = new EmployeeService();

        this.find = this.find.bind(this);

        this.findAll = this.findAll.bind(this);

    }

    async findAll(request: Request, response: Response): Promise<Employee[] | Response> {

        try {

            const employees:Employee[] = await this.employeeService.findAll();

            return response.status(200).json(employees);

        } catch (error) {

            console.log(error);

            return response.status(500).json(error.message);

        }

    }

    async find(request: Request, response: Response): Promise<Employee[] | Response> {

        try {

            const employees: Employee[] = await this.employeeService.find(parseInt(request.params.pageNo), parseInt(request.params.limit));

            return response.status(200).json(employees);

        } catch (error) {

            console.log(error);

            return response.status(500).json(error.message);

        }

    }

}

**Create Routing**

In **routes** directory create a routing file for employee controller **employeeroutes.ts**

Content of **employeeroutes.ts**

import { EmployeeController } from "../controllers/EmployeeController";

import { Router } from "express";

const employeeRouter = Router();

const employeeController:EmployeeController=new EmployeeController();

employeeRouter.get('/', employeeController.findAll);

employeeRouter.get('/page/:pageNo/limit/:limit', employeeController.find);

export default employeeRouter;

Now get back to index.ts file and use routes there

import express from 'express';

import employeeRouter from './src/routes/employeeroutes';

const server = express();

server.use(express.json());

server.use(express.urlencoded({ extended: true }));

const PORT = 9000;

server.use('/api/employees',employeeRouter);

server.get('/', (req, res) => res.send('Express + TypeScript Server'));

server.listen(PORT, () => {

    console.log(`⚡️[server]: Server is running at http://localhost:${PORT}`);

});

Now open the terminal and execute following command to run the project:

**npm start**

Test these APIs on postman

**Dependency Injection**

**InversifyJS** is a lightweight inversion of control (IoC) container for TypeScript and JavaScript apps. An IoC container uses a class constructor to identify and inject its dependencies. InversifyJS has a friendly API and encourages the usage of the best OOP and IoC practices.

Download with npm inversifyjs  
**npm install inversify**

In our case EmployeeController is dependent on the object of EmployeeService and the object of EmployeeService is dependent on the object of EmployeeDao. We will inject these dependencies with the help of IOC Container provided by inversify. So, for that purpose we have to declare them **injectable** and when we need these dependencies we will simply **inject** them.  
  
inversifyJs provides some decorators like **@injectable** and **@inject**

⚠️ **Important!** InversifyJS requires TypeScript >= 2.0 and the **experimentalDecorators option set to true** in your **tsconfig.json** file.

InversifyJS requires a modern JavaScript engine with support for:

* [Reflect metadata](https://rbuckton.github.io/reflect-metadata/)
* [Map](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map)

We also need to install a library which is required for dependency injection: **reflect-metadata**

**npm install reflect-metadata**

let us start by making our dao as injectable. Just place a decorator @injectable on dao to make it injectable into another classes:

import {Employee} from '../models/Employee';

import { DbConfig } from '../config/database';

import knex from "knex";

import { injectable } from 'inversify';

const db=knex(DbConfig);

@injectable()

export class EmployeeDao{

    async findAll(): Promise<Employee[]>{

        try {

            const employees:Employee[] =await db<Employee>('employees').select();

            return employees;

        } catch (error) {

            throw error;

        }

    }

    async find(limit: number,offset: number): Promise<Employee[]>{

        try {

            const employees:Employee[]= await db<Employee>('employees').select().limit(limit).offset(offset);

            return employees;

        } catch (error) {

            throw error;

        }

    }

}

Now we will inject the EmployeeDao dependency in EmployeeService and make EmployeeService injectable in order to inject it into EmployeeController.  
To inject the dependency, use **@inject** decorator and in the parameter we pass the symbol that will be used to identify the dependency which we want to inject.

Define the symbols for the dependencies:  
Create a folder named **types** in **src** directory. In **types** folder create another folder **DependencyInjectorSymbols .** In this folder create a file symbols.ts

Content of symbols.ts:

const TYPES = {

    EmployeeService: Symbol('EmployeeService'),

    EmployeeDao: Symbol('EmployeeDao'),

    EmployeeController: Symbol('EmployeeController'),

};

export default TYPES;

( this is similar to java where we assign a name to a bean. This name is used to identify the bean. Here in inversify, instead of name we are assigning symbol to identify them )

Now update EmployeeService.ts as follows:

import { inject, injectable } from "inversify";

import { EmployeeDao } from "../dao/EmployeeDao";

import { Employee } from "../models/Employee";

import TYPES from "../types/DependencyInjectorSymbols/symbols";

@injectable()

export class EmployeeService{

    private employeeDao: EmployeeDao;

    constructor(@inject(TYPES.EmployeeDao) employeeDao: EmployeeDao) {

        this.employeeDao = employeeDao;

    }

    async findAll(): Promise<Employee[]> {

        try {

            const employees:Employee[] = await this.employeeDao.findAll();

            return employees;

        } catch (error) {

            throw error;

        }

    }

    async find(pageNo: number, limit: number): Promise<Employee[]> {

        try {

            const offset:number = (pageNo - 1) \* limit;

            const employees:Employee[] = await this.employeeDao.find(limit, offset);

            return employees;

        } catch (error) {

            throw error;

        }

    }

}

Here we are injecting the employeeDao as dependency by telling @inject that inject the dependency mapped by TYPES.EmployeeDao symbol which was declared in symbols.ts and later on, we will configure that actually whose object we need to inject for each symbol.

Now inject the EmployeeService dependency in EmployeeController.   
Update EmployeeController.ts as follows:

import { Request, Response } from 'express';

import { inject, injectable } from 'inversify';

import { Employee } from '../models/Employee';

import { EmployeeService } from '../services/EmployeeService';

import TYPES from '../types/DependencyInjectorSymbols/symbols';

@injectable()

export class EmployeeController {

    private employeeService: EmployeeService;

    constructor(@inject(TYPES.EmployeeService) employeeService: EmployeeService) {

        this.employeeService = employeeService;

        this.find = this.find.bind(this);

        this.findAll = this.findAll.bind(this);

    }

    async findAll(request: Request, response: Response): Promise<Employee[] | Response> {

        try {

            const employees:Employee[] = await this.employeeService.findAll();

            return response.status(200).json(employees);

        } catch (error) {

            console.log(error);

            return response.status(500).json(error.message);

        }

    }

    async find(request: Request, response: Response): Promise<Employee[] | Response> {

        try {

            const employees: Employee[] = await this.employeeService.find(parseInt(request.params.pageNo), parseInt(request.params.limit));

            return response.status(200).json(employees);

        } catch (error) {

            console.log(error);

            return response.status(500).json(error.message);

        }

    }

}

Now we will configure the inversify ioc container that for which symbol we need to inject whose object. For that purpose create a file inversify.config.ts in config directory

Content of inversify.config.ts

import { Container } from 'inversify';

import { EmployeeDao } from '../dao/EmployeeDao';

import { EmployeeService } from '../services/EmployeeService';

import TYPES from '../types/DependencyInjectorSymbols/symbols';

const container: Container = new Container();

container.bind<EmployeeService>(TYPES.EmployeeService).to(EmployeeService).inSingletonScope();

container.bind<EmployeeDao>(TYPES.EmployeeDao).to(EmployeeDao).inSingletonScope();

export default container;

here you can clearly observe that we have binded each symbol to the particular dependency.

Also update the content of employeeroutes.ts

import { EmployeeController } from "../controllers/EmployeeController";

import { Router } from "express";

import container from "../config/inversify.config";

const employeeRouter = Router();

**const employeeController: EmployeeController = container.resolve<EmployeeController>(EmployeeController);**

employeeRouter.get('/', employeeController.findAll);

employeeRouter.get('/page/:pageNo/limit/:limit', employeeController.find);

export default employeeRouter;

now at last import the reflect-metadata library in index.ts file

import 'reflect-metadata';

import express from 'express';

import employeeRouter from './src/routes/employeeroutes';

const server = express();

server.use(express.json());

server.use(express.urlencoded({ extended: true }));

const PORT = 8000;

server.use('/api/employees', employeeRouter);

server.get('/', (req, res) => res.send('Express + TypeScript Server'));

server.listen(PORT, () => {

    console.log(`⚡️[server]: Server is running at http://localhost:${PORT}`);

});

Now run the application by command: **npm start**